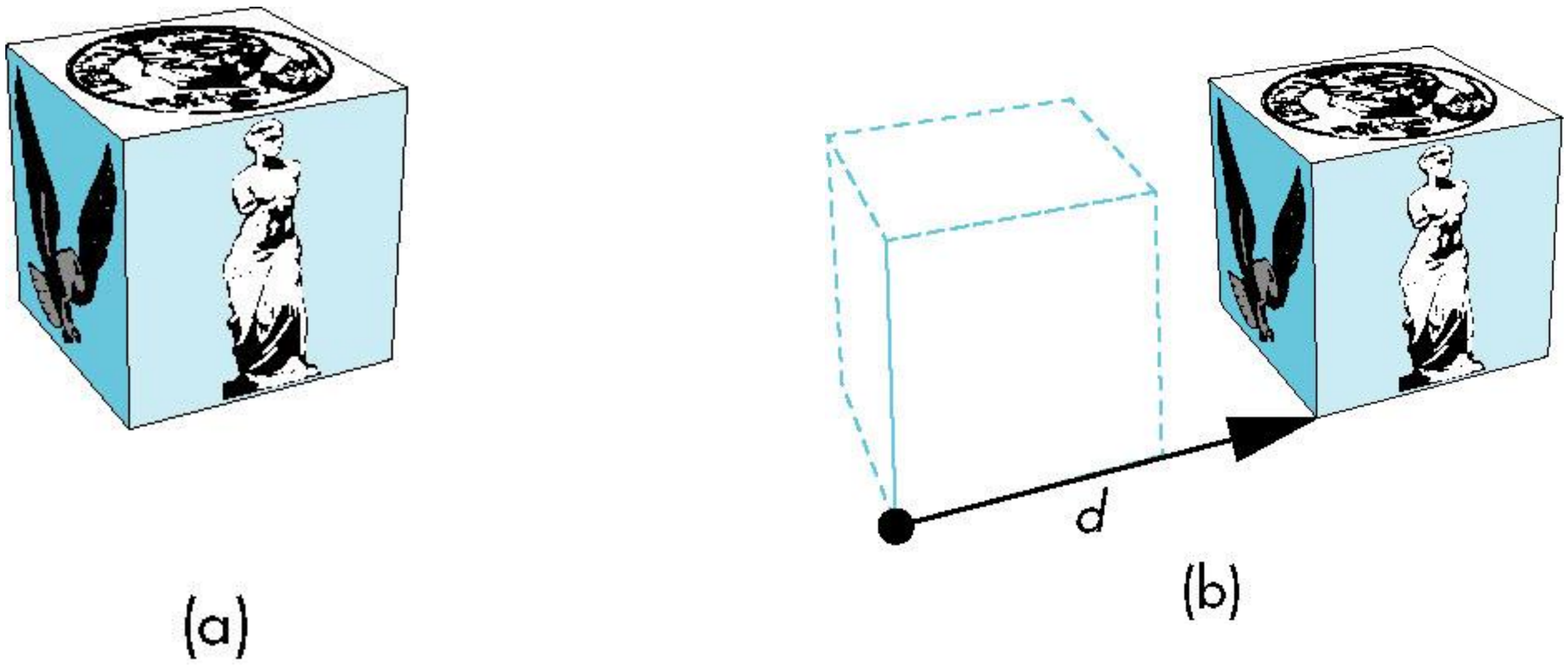
Lecture 12

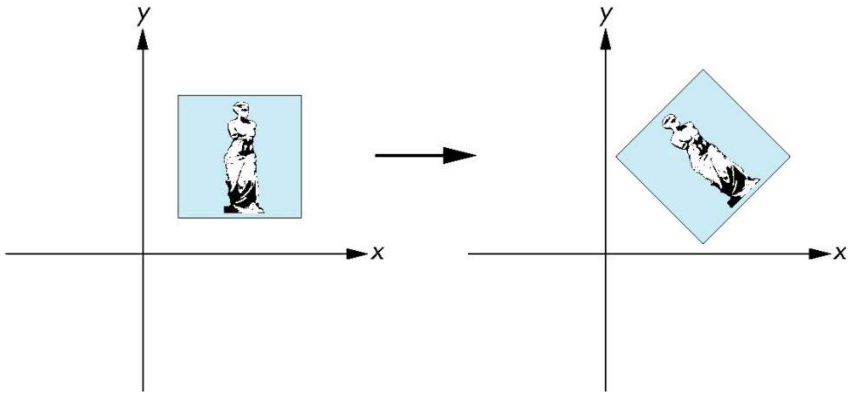
A rotating colored cube

Affine translation explained

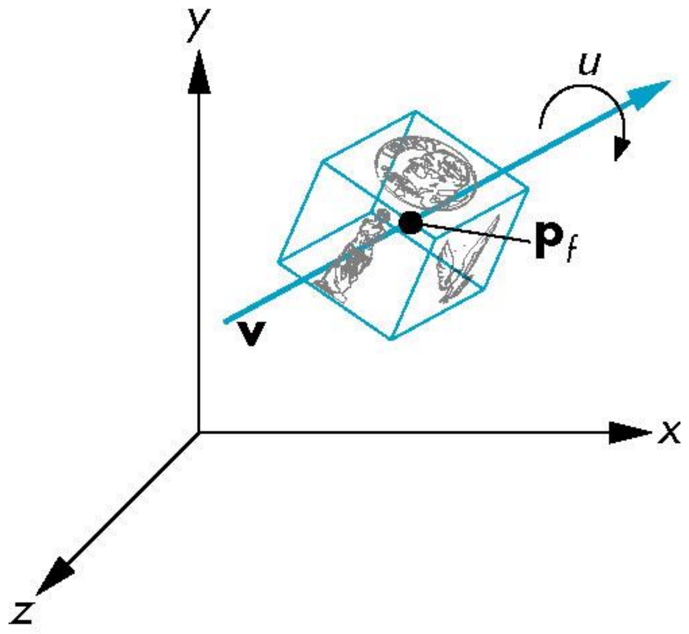


(a) Object before translation, (b) object after translation

Affine Rotation explained

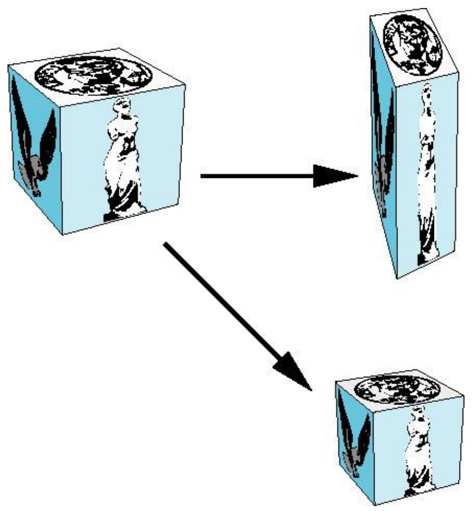


2D Rotation around a fixed point (object center)

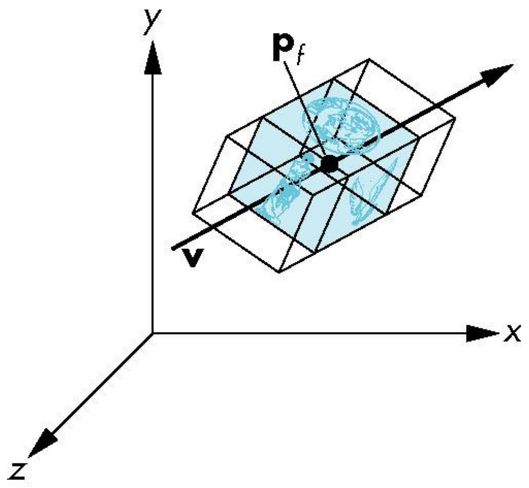


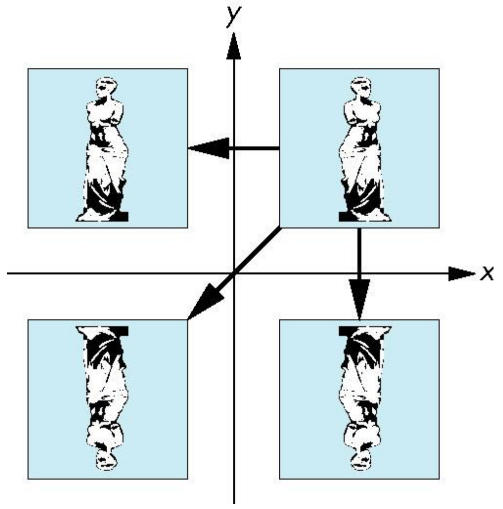
3D Rotation around a vector

Affine Scaling explained



Uniform scaling(the same scaling in all direction) in right-down image. Non uniform scaling the upper-right image





- The scaling is done around a fixed point. A scaling factor α is specified for each of the three direction in 3D (left image).
    - α >1 means enlarging
    - α <1 means shrinking
- Negative α means enlarging (α>1) or shrinking (α<1) both with reflection about the fixed point (right image)

Affine (line preserving) Transformation summary

Transformation: Given a object representation in a given frame, what is the representation in the same frame if the object is moved, rotated, scaled, sheared, etc.

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & \alpha_x \\ 0 & 1 & 0 & \alpha_y \\ 0 & 0 & 1 & \alpha_z \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

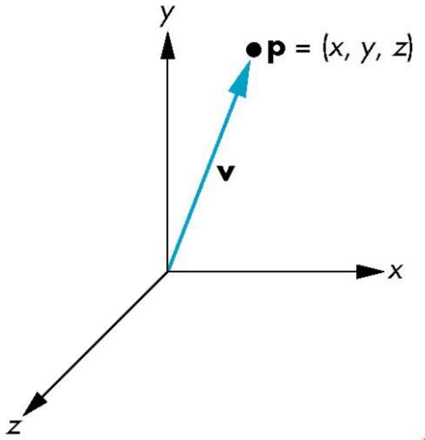
$$\mathbf{T}^{-1}(\alpha_x, \alpha_y, \alpha_z) = \mathbf{T}(-\alpha_x, -\alpha_y, -\alpha_z)$$

Translation

$$\mathbf{S} = \mathbf{S}(\beta_x, \beta_y, \beta_z) = \begin{bmatrix} \beta_x & 0 & 0 & 0 \\ 0 & \beta_y & 0 & 0 \\ 0 & 0 & \beta_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

$$\mathbf{S}^{-1}(\beta_x, \beta_y, \beta_z) = \mathbf{S}\left(\frac{1}{\beta_x}, \frac{1}{\beta_y}, \frac{1}{\beta_z}\right)$$

Scaling



$$\mathbf{R}_z = \mathbf{R}_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

$$\mathbf{R}_x = \mathbf{R}_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

$$\mathbf{R}_y = \mathbf{R}_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

$$\mathbf{R}^{-1}(\theta) = \mathbf{R}(-\theta). \qquad \mathbf{R}^{-1}(\theta) = \mathbf{R}^T(\theta).$$

Rotation

Shear

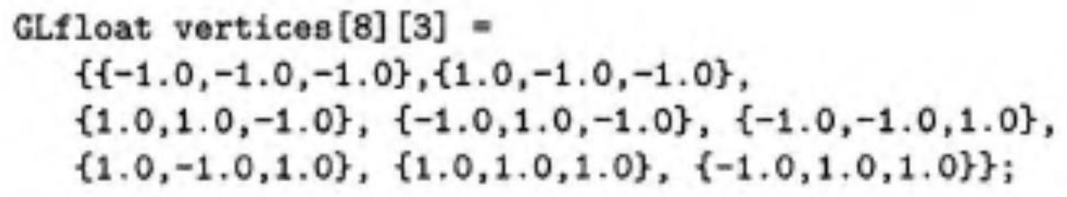
$$\mathbf{H}_x(\theta) = \begin{bmatrix} 1 & \cot\theta & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{H}_x^{-1}(\theta) = \mathbf{H}_x(-\theta)$$

Affine transformation matrices follow the vector data layout in matrix form and multiplication order in OpenGL

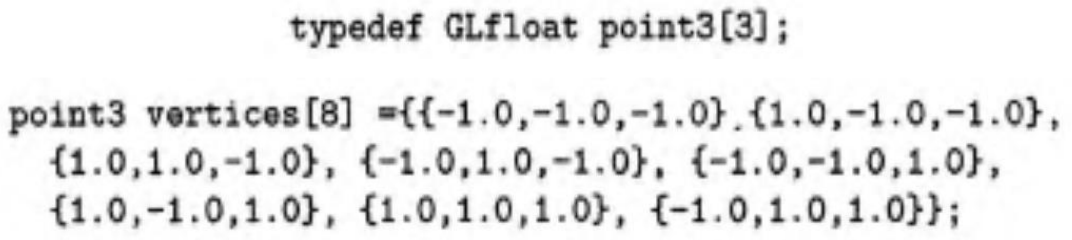
Modeling the cube faces

Modeling the 8 vertices
Using the object coordinates



```
GLfloat vertices[8][3] =
   {{-1.0,-1.0,-1.0},{1.0,-1.0,-1.0},
   {1.0,1.0,-1.0}, {-1.0,1.0,-1.0}, {-1.0,-1.0,1.0},
   {1.0,-1.0,1.0}, {1.0,1.0,1.0}, {-1.0,1.0,1.0}};
```

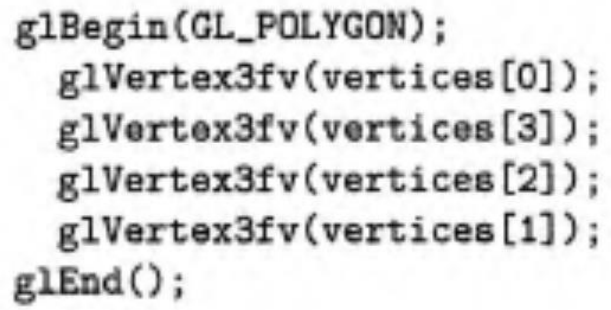
OR



```
            typedef GLfloat point3[3];

point3 vertices[8] ={{-1.0,-1.0,-1.0},{1.0,-1.0,-1.0},
  {1.0,1.0,-1.0}, {-1.0,1.0,-1.0}, {-1.0,-1.0,1.0},
  {1.0,-1.0,1.0}, {1.0,1.0,1.0}, {-1.0,1.0,1.0}};
```

Modeling one face as an example. We still working in the object coordinates

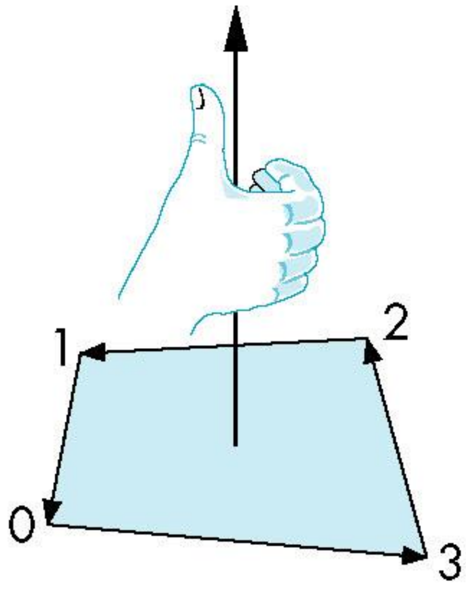


```
glBegin(GL_POLYGON);
   glVertex3fv(vertices[0]);
   glVertex3fv(vertices[3]);
   glVertex3fv(vertices[2]);
   glVertex3fv(vertices[1]);
glEnd();
```

Inward- and outward-Pointing faces

Each polygon has two sides. The graphics system can display either or both of them. The order in which the vertices are specified determine which is the outward face (the other is the inward face)

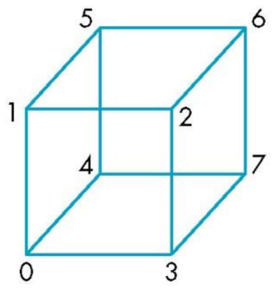
The right-hand rule is applied to determine the outward face as shown
or
If you are looking to the face (the face is right in front of you) and the vertices are specified in counterclockwise, then you are looking to the outward face

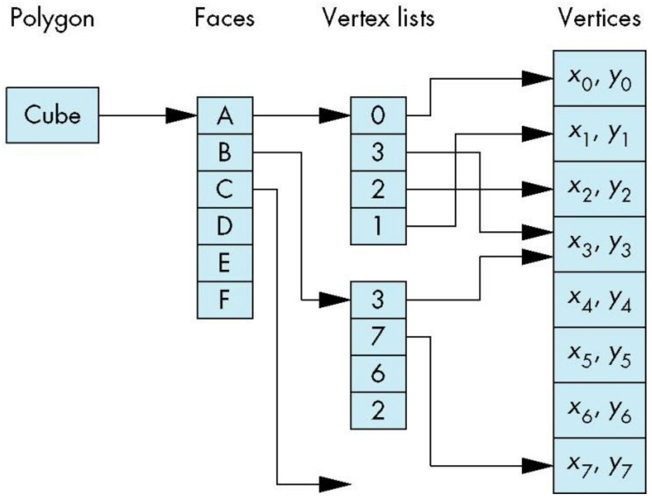


By specifying outward (front) and inward (back) carefully, we will be able to eliminate (or cull) faces that are not visible or to use different attributes to display front and back faces.

Modeling a cube in concentrating on the topology(structure) rather than geometry

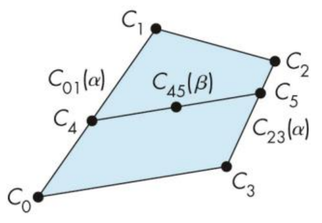
- Here the top-level entity is a cube that is assembled from faces entities. This topology structure is valid for any cube
- The data of the cube which makes one cube differs from another is stored in the order of vertices for each face and the geometry(locations) specified by the numbers stored in the vertices list
- This is just an example to model a cube . We can use any other model that separates the data from the structure(class in a high-level language). This is the OO way to describe entities
- The idea is that each geometric(data) location appears just once in the model (imagine that you model a cube by directly listing the vertices in polygons where each vertex will be repeated three times in the model

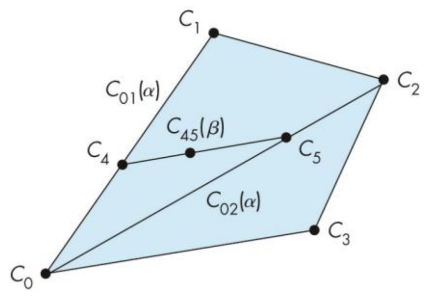




Modeling the colored cube

- Here, the color used for each vertex is a solid color. The color of any point inside the face is the interpolation of the color of the point w.r.t. the color of the face vertices
- Many interpolation methods could be used to color the face using the vertices colors(the figures shows bilinear)
- We can instruct OpenGL to use our preferred way to interpolate





```
GLfoat vertices[8][3] = {{-1.0,-1.0,1.0},{-1.0,1.0,1.0},
    {1.0,1.0,1.0}, {1.0,-1.0,1.0}, {-1.0,-1.0,-1.0},
    {-1.0,1.0,-1.0}, {1.0,1.0,-1.0}, {1.0,-1.0,-1.0}};

GLfloat colors[8][3] = {{0.0,0.0,0.0},{1.0,0.0,0.0},
    {1.0,1.0,0.0}, {0.0,1.0,0.0}, {0.0,0.0,1.0},
    {1.0,0.0,1.0}, {1.0,1.0,1.0}, {0.0,1.0,1.0}};

void quad(int a, int b, int c, int d)
{
  glBegin(GL_QUADS);
      glColor3fv(colors[a]);
      glVertex3fv(vertices[a]);
      glColor3fv(colors[b]);
      glVertex3fv(vertices[b]);
      glColor3fv(colors[c]);
      glVertex3fv(vertices[c]);
      glColor3fv(colors[d]);
      glVertex3fv(vertices[d]);
  glEnd();
}

void colorcube()
{
    quad(0,3,2,1);
    quad(2,3,7,6);
    quad(0,4,7,3);
    quad(1,2,6,5);
    quad(4,5,6,7);
    quad(0,1,5,4);
}
```

The rotating colored cube

```
glutDisplayFunc(display);
glutIdleFunc(spincube);
glutMouseFunc(mouse);
```

```
void display()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();
    glRotatef(theta[0], 1.0, 0.0, 0.0);
    glRotatef(theta[1], 0.0, 1.0, 0.0);
    glRotatef(theta[2], 0.0, 0.0, 1.0);
    colorcube();
    glutSwapBuffers();
}
```

```
void mouse(int btn, int state, int x, int y)
{
    if(btn==GLUT_LEFT_BUTTON && state == GLUT_DOWN) axis = 0;
    if(btn==GLUT_MIDDLE_BUTTON && state == GLUT_DOWN) axis = 1;
    if(btn==GLUT_RIGHT_BUTTON && state == GLUT_DOWN) axis = 2;
}
```

```
void spinCube()
{
   theta[axis] += 2.0;
   if( theta[axis] > 360.0 ) theta[axis] -= 360.0;
   glutPostRedisplay();
}
```

```
void mykey(char key, int mousex, int mousey)
{
    if(key=='q'||key=='Q') exit();
}
```

The Rotating Colored Cube(RCC) Application

Run RCC

Note, if you see visual artifacts, just change the waiting time in the program to adjust acceding to the speed of you hardware

Rotation by changing the camera

- In this example, We showed a moving object in front of a fixed camera position and orientation.
- The rendered image on the screen can be changed also due to a moving/changing orientation camera in a fixes static scene
- In our example, we could have a similar result if we let the cube fixed in its place and moving the camera along a circle around it, at each location on the circle the camera is always directed towards the location of the cube